Polynomial time reduction and NP-completeness

Exploring some time complexity limits of polynomial time algorithmic solutions
Polynomial time reduction

**Definition**: A language $L$ is said to be *polynomial time reducible* or *reducible in polynomial time* to a language $U$ if the map reduction between them can be computed in polynomial time.

- This is, the time complexity of the Turing machine that computes the reduction mapping is bounded above by a polynomial function in the length of the input string.
Example of polynomial time reducible languages

**Definition**: A clause is a Boolean formula consisting of a set of Boolean variables connected by the logical or. A Boolean formula is said to be in **conjunctive normal form** (CNF) if it consists of clauses connected by the logical and.

**Notations**:
- 3CNF denotes the set of all conjunctive normal forms whose clauses involve at most 3 Boolean variables
- 3SAT denotes the set of all satisfiable Boolean formulas in 3CNF (Thus, 3SAT is a subset of 3CNF)
Example (cont.)

**Definition**: An $n$-clique in an undirected graph is a fully connected (this is, all-to-all) $n$-node sub-graph of the graph. $nClique$ denotes the set of all undirected graphs possessing an $n$-clique.

**Theorem 20.1**: $3SAT$ is reducible to $nClique$ in polynomial time.
Example (cont.)

Proof: Let

$3CNF = \{ \phi : \phi \text{ is a Boolean formula } \land (\exists n \geq 0) \land \phi = (x_1 \lor x_2 \lor x_3) \land \cdots \land (x_{3n+1} \lor x_{3n+2} \lor x_{3n+3}) \}$

where $n$ is, from now on, the number of clauses in $\phi$.

$3SAT = \{ \phi : \phi \in 3CNF \land \phi \text{ is satisfiable} \}$

$Graphs = \{ G : G \text{ is an undirected graph} \}$

$nCLIQUE = \{ G : G \text{ is an undirected graph possessing a } n \text{- clique} \}$

Define:

$f : 3CNF \rightarrow Graphs$

$f(\phi) = G$, where $G = (\{x_1, \ldots, x_{3n+3}\}, E)$,

and $E = \{ \{ x_{3i+j}, x_{3k+s} \} : 1 \leq j, s \leq 3 \land (\forall i, k)(0 \leq i, k \leq n \land i \neq k \land x_{3k+s} \neq \neg x_{3i+j}) \}$
Example (cont.)

Sub-claim 1: \( f \) is a reduction from \( n\text{Clique} \) to 3SAT

\[ \phi \in 3\text{SAT} \Rightarrow \text{Each clause in } \phi \text{ is satisfied} \]

\[ \Rightarrow \text{At least one literal is true in each clause} \]

\[ \Rightarrow (\exists i_1, \ldots, i_n, j_1, \ldots, j_n) \land (\forall l, k)(l \neq k \Rightarrow i_l \neq i_k) \land (\forall l)x_{3i_l+j_l} = 1 \]

\[ \Rightarrow (\forall l, k)(l \neq k \Rightarrow x_{3i_l+j_l} \neq \neg x_{3i_k+j_k}) \]

\[ \Rightarrow \{x_{i_k+j_k} : k = 1, \ldots, n\} \text{ is an } n-\text{clique in } G \]

\[ \Rightarrow f(\phi) \in n\text{CLIQUE} \]

\[ f(\phi) \in n\text{CLIQUE} \Rightarrow (\exists \text{ an } n-\text{clique } \{y_1, \ldots, y_n\} \text{ in } f(\phi)) \]

\[ \Rightarrow (\exists i_1, \ldots, i_n, j_1, \ldots, j_n) \land y_k = x_{3i_k+j_k} \land (\forall k, l)y_k \neq \neg y_l \]

\[ \Rightarrow (\forall k)(y_k = x_{3i_k+j_k} = 1 \Rightarrow \phi = 1) \]

\[ \Rightarrow \phi \in 3\text{SAT} \]
Example (cont.)

Sub-Claim 2: \( f \) is computable in polynomial time
The following Turing machine computes \( f \)
F="On input \( w \):
    Sweep across the string verifying that \( w \) is in 3CNF.
    If \( w \) is not in 3CNF, REJECT.
    Else,
    Write the set of all literals in \( w \) on tape.
    For each clause \( C \) in \( w \),
        For each literal \( a \) in \( C \),
            For each literal \( b \) in \( w \) but not in \( C \)
                If \( b \neq \neg a \) write \( \{a, b\} \) on tape
        HALT."
The time complexity of \( F \) is bounded above by \( |w|^3 \)
The class NP

**Definition**: A language (problem) \( L \) is said to be \( \text{NP} \) (or to be in the class \( \text{NP} \)) if its best known time complexity bound is achieved by a **nondeterministic decider** which, on each input string produces either a polynomial time accepting branch, or its longest rejection branch is polynomially bounded.

- **Observation**: A direct application of Theorem 18.2 lead us to the conclusion that the transformation of the nondeterministic TM to a deterministic TM will render exponential time complexity.
NP is often misused

NP stands for **Nondeterministic Polynomial time**

and not for **Non-polynomial time**

– Non-polynomial time conveys a larger class of problems, including those without any known NP solution
Verifying $NP$ solutions

When it comes to recognizing a string, the difference between $P$ and $NP$ is in the overhead imposed by searching for the appropriate accepting computation branch. Indeed, a string in an $NP$ language is accepted in polynomial time whenever the computation starts in the polynomial time accepting branch

– This idea is at the base of the notion of polynomial time verifiers
Verifiers

**Definition**: A **verifier** for a language $L$ is a **decider** $V$ so that $L$ can be written as

$$L = \{ w \in A^*: (\exists c) \land V \text{ accepts } < w, c > \}$$

Thus, a **verifier** is a **decider** used for expressing membership in $L$. Such expression acquires the form of the acceptance of a pair of strings, one representing a problem’s instance and the other, its “solution”
Showing that a language is NP

Verifiers provide a simpler way to demonstrate that a given language belongs to NP:

In general, showing that a language is NP amounts to proving that:

a) The language is **decidable**

b) There is a **polynomial time verifier** for the language, this is, the verifier halts in polynomial time
Theorem 20.2 : SAT is \textit{NP}

Proof: SAT is decidable (Theorem 19.5). The next TM is a verifier for SAT:

\textit{V}="On input \(<w, c>\), where \(w\) is a string representing a Boolean formula and \(c\) a string in \(\{0, 1\}^*\), \(|w| = |c|:\)
\begin{enumerate}
\item If \(|w| = |c|\)
   \begin{enumerate}
   \item "Evaluate" \(w\) in \(c\)
   \item If \(w = 1\), ACCEPT. If \(w = 0\) REJECT.
   \end{enumerate}
\item Else, REJECT."
\end{enumerate}

Clearly \(V\) is \(O(|w|)\), and a verifier of SAT since:

\[
SAT = \{w : (\exists c \in \{0,1\}^*) \land |c| = |w| \land V \text{ accepts } <w, c>\}
\]
A non-NP problem

\( k \)CLIQUECOMP is described as follows: Let \( UG \) be the set of all undirected graphs and \( N \) the natural numbers

**Problem domain**: \( UG \times N \)

**Problem instance**: \( G \) in \( UG \), and \( k \) in \( N \)

**Question**: Is it true that \( G \) has no \( k \)-cliques?

– Verifying that \( G \) has no \( k \)-cliques by exhaustion requires checking an exponentially large number of possibilities. But exhaustion is the only known method for accomplishing this task. Thus, no polynomial time verifier is known for \( k \)CLIQUECOMP
NP-completeness, NP-hardness

**Definition:** A problem (language) $L$ is said to be **NP-complete** if
a) The problem (language) $L$ is NP; and
b) Every NP problem (language) is reducible to $L$ in polynomial time

**Definition:** A problem (language) $L$ is said to be **NP-hard** if
a) Every NP problem (language) is reducible to $L$ in polynomial time
P=NP? a long-standing open problem

It is clear from their definitions that P is a subclass of NP and that NP-complete is a subclass of NP-hard

The question of whether the class P is a strict subset of the class NP or an equality disguised as a subset remains unanswered

– It is claimed that an answer to this question has practical consequences. Indeed, it all depends, as polynomial algorithms of time complexity of the order $O(n^k)$, $k \geq 3$ are often as impractical as exponential time methods
$NP$-complete and the $P=NP$ question

Cook, Levin, Karp are among the many theoretical computer scientists that have attempted to answer whether $P=NP$

– $NP$-complete languages were introduced by Cook in the 70’s in the midst of these theoretical efforts
The Cook-Levin Theorem

**Theorem 20.3**: SAT is NP-complete

**Proof**: Let $L$ be an NP language, and $D$ a decider for $L$. Then, for each $w$ in $L$, there is a polynomial time computation path induced by $D$. The size of the tableau described in the proof of Theorem 19.4 is then, bounded by $O(|w|^n)$, for some $n > 0$. Therefore, the TM Phi in the proof of the same theorem, construct the Boolean formula associated with the tableau in polynomial time, as well.
Cook-Levin Theorem (cont.)

Let $O(|w|^k)$ be the time complexity of $M$. Then, the time complexity of Phi is estimated as follows:

Step 1: An accepting run of $M$ takes $O(|w|^k)$ time. And writing the tableau takes

$$O(|w|^k)O(|w|^k) = O(|w|^{2k})$$
time. Therefore, step 1 is $O(|w|^{2k})$

Step 2: Writing each of $\phi_{cell}(w), \phi_{move}(w), \phi_{acc}(w)$ demands inspecting the whole tableau, and therefore, the writing takes $O(|w|^{2k})$;

$\phi_{start}(w)$ is written in $O(|w|^k)$. 
More theory

Theorem 20.4: If $L$ is polynomial time reducible to $U$ and $U$ is in $P$, then $L$ is in $P$.

Proof: Let $M$ be a polynomial time decider for $U$ and $f$ a polynomial time reduction map from $L$ to $U$. Construct:

$N$="On input $w$:
1. Compute $f(w)$.
2. Run $M$ on $f(w)$ and output whatever $M$ outputs."

Clearly, $N$ decides $L$ in polynomial time.
The P=NP connection

The previous result provides the connection between NP-completeness and the P=NP question. By virtue of the Theorem 20.2, if a polynomial time TM is found for solving an NP-complete problem then P = NP.

Unfortunately, no such TM has been discovered, so far…
More theoretical facts

Theorem 20.5: If \( f \) and \( g \) are polynomial time computable mappings, and the composition \( f \circ g \) is well-defined, then \( f \circ g \) is computable in polynomial time.

Proof: Let \( F \) and \( G \) be the TMs that compute \( f \) and \( g \), respectively. Define:

\[ C = \text{"On input } w: \]
\[ \quad \text{Run } F \text{ on } w \]
\[ \quad \text{When } F \text{ halts, run } G \text{ on } f(w) \]
\[ \quad \text{Halt"} \]

The time complexity of \( C \) is bounded \( O(F)O(G) \) which is polynomial.
More theoretical facts (cont.)

Theorem 20.6: If $L$ is polynomial time reducible to $U$, $U$ is $NP$ and $L$ is in $NP$-complete, then $U$ is $NP$-complete

Proof: Let $f$ be a polynomial time reduction from $L$ to $U$ and let $Z$ be an $NP$ language. Since $L$ is $NP$-complete, there is $f$, polynomial time reduction from $Z$ to $L$. Let $g$ be the polynomial time reduction from $Z$ to $L$. Then $f \circ g$ is a polynomial time reduction from $Z$ to $U$. Since, by hypothesis $U$ is $NP$, $U$ is $NP$-complete
More theoretical facts (cont.)

Theorem 20.7: If $L$ is polynomial time reducible to $U$, and $L$ is in $NP$-hard, then $U$ is in $NP$-hard

Proof: Let $f$ be a polynomial time reduction from $L$ to $U$ and let $Z$ be an $NP$ language. Since $L$ is $NP$-hard, there is $f$, polynomial time reduction from $Z$ to $L$. Let $g$ be the polynomial time reduction from $L$ to $U$. Then $f \circ g$ is a polynomial time reduction from $Z$ to $U$. Therefore, $U$ is $NP$-hard
Reduction as a tool

The previous theorems are extensively used for determining the complexity class of a problem. In particular, SAT is often used as a reference for NP-complete problems in the sense that:

If a problem is NP and SAT is polynomially reducible to the problem, then the problem itself is NP-complete.

A similar technique works for NP-hard problems.